

Chapter 1

Introduction

The *Unified Modeling Language* (UML) is a consolidation of the best practices that have been established over the years in the use of modeling languages. UML enables us to present the widely varying aspects of a software system (e.g., requirements, data structures, data flows, and information flows) within a single framework using object-oriented concepts. Before we venture too deeply into UML, however, in this chapter we first explain why modeling is an indispensable part of software development. To do this, we look at what models are and what we need them for. We briefly recapitulate the basic concepts of object orientation before continuing with an overview of the structure of the book.

*Unified Modeling
Language (UML)*

1.1 Motivation

Imagine that you want to develop a software system that a customer has ordered from you. One of the first challenges you are confronted with is clarifying what the customer actually wants and whether you have understood the customer's exact requirements for the prospective system. This first step is already critical for the success or failure of your project. The question is, how do you communicate with your customer? Natural language is not necessarily a good choice as it is imprecise and ambiguous. Misunderstandings can easily arise and there is a very great risk that people with different backgrounds (e.g., a computer scientist and a business manager) will talk at cross-purposes, which can have serious consequences.

What you need is to be able to create a model for your software. This model highlights the important aspects of the software in a clear form

of notation that is as simple as possible but abstracts from irrelevant details, just like models in architecture, e.g., construction plans. A construction plan for a building contains information such as the floor plan. Construction materials to be used are not specified at this point in time; they are irrelevant and would make the plan more complicated than necessary. The plan also does not contain any information about how the electrical cables are to be laid. A separate plan is created for this aspect to avoid presenting too much information at once. Just like in architecture, it is important in information technology that people with different backgrounds (e.g., architect and builder) can read, interpret, and implement the model.

Modeling language

Modeling languages were developed precisely for such scenarios and demonstrate clearly defined rules for a structured description of a system. These languages can be *textual* (e.g., a programming language such as Java) or *visual* (e.g., a language that provides symbols for transistors, diodes, etc. that can be combined with one another). Modeling languages can be designed for a specific domain, for example, for describing web applications. On the one hand, these *domain-specific modeling languages* provide tools and guidelines for solving problems in a specific field efficiently; on the other hand, they can also be restrictive. Alternatively, modeling languages can be designed for general purpose use. The language UML, which is the subject of this book, is a general purpose modeling language. We will use UML to get to know the most important concepts of object-oriented modeling.

Object-oriented modeling

Object-oriented modeling is a form of modeling that obeys the object-oriented paradigm. In the following two subsections, we will look briefly at the notion of a model and the main concepts of object orientation. This will provide us with a good basis for our subsequent examination of object-oriented modeling with UML.

1.2 Models

System

Models allow us to describe systems efficiently and elegantly. A *system* is an integrated whole made up of components that are related to one another and influence each other in such a way that they can be perceived as a single, task-based or purpose-based unit. In this regard, they separate themselves from the surrounding environment [52]. Examples of systems are material things, such as cars or airplanes, ecological environments, such as lakes and forests, but also organizational units such as a university or a company. In information technology, we are interested in particular in software systems and thus in models that describe software systems.

Software system

Software systems themselves are based on *abstractions* that represent machine-processible facts of reality. In this context, abstraction means generalization—setting aside specific and individual features. Abstract is the opposite of concrete. Abstracting therefore means moving away from specifics, distinguishing the substance from the incidental, recognizing common characteristics [29].

Abstraction

When creating software systems, it is extremely important to select suitable means of abstraction: on the one hand for the implementation, but on the other hand also for the subsequent use of the software systems. Choosing the correct means of abstraction makes programming easier. The individual parts then have simple and small interfaces. New functionality can be introduced without the need for extensive reorganization. Choosing the wrong means of abstraction might result in a number of nasty surprises during implementation: the interfaces will be complicated and it will be difficult to implement changes. You can only manage the ever-increasing complexity of modern software systems with suitable means of abstraction [26]. This is where modeling can provide valuable services.

Selecting means of abstraction

To develop a better understanding of modeling concepts, below we present widespread and generally recognized definitions of the notion of a model as well as the properties that a good model should possess.

The notion of a *model* is important not only in information technology but also in many other scientific disciplines (mathematics, philosophy, psychology, economics, etc.). Derived from the Latin “modulus”, which designates a scale in architecture, during the Renaissance the word “modello” was used in Italy for an illustrative object intended to present the form and design of a planned building to a client and to clarify design and architectural questions. Over the subsequent centuries, the notion of a “model” has been used in various branches of science for a simplified description of complex facts from reality.

Model

In 1973, Herbert Stachowiak proposed a model theory that is distinguished by three characteristics [48]:

Definition by Herbert Stachowiak

1. *Mapping*: a model is always an image (mapping) of something, a representation of natural or artificial originals that can be models themselves.
2. *Reduction*: a model does not capture all attributes of the original, rather only those that seem relevant to the modeler or user of the model.
3. *Pragmatism*: pragmatism means orientation toward usefulness. A model is assigned to an original based on the following questions: *For whom? Why? What for?* A model is used by the modeler or user instead of the original within a specific time frame and for a specific purpose.

Models support a representation of a system that is reduced to the essentials in order to minimize the complexity of the system to manageable aspects. A system is usually described not by one single view but by a number of views that together produce a unified overall picture. Thus, one view might describe the objects involved and their relationship to one another; another view might describe the behavior of a group of objects or present the interactions between different objects.

Properties of models

Models must be created with great care and due consideration. According to Bran Selic [47], five characteristics determine the quality of a model:

- *Abstraction*: a model is always a reduced representation of the system that it represents. Because the details that are irrelevant in a specific context are hidden or removed, it is easier for the user to understand the essence of the whole.
- *Understandability*: simply omitting irrelevant details is not enough to make a model understandable. It is important to present the remaining elements as intuitively as possible—for example, in a graphical notation. The understandability results directly from the expressiveness of the modeling language. Expressiveness can be defined as the ability to present complex content with as few concepts as possible. In this way, a good model reduces the intellectual effort required to understand the content depicted. For example, typical programming languages are not particularly expressive for a human reader as a lot of effort is required to understand the content of the program.
- *Accuracy*: a model must highlight the relevant properties of the real system, reflecting reality as closely as possible.
- *Predictiveness*: a model must enable prediction of interesting but not obvious properties of the system being modeled. This can be done via simulation or analysis of formal properties.
- *Cost-effectiveness*: in the long-run, it must be cheaper to create the model than to create the system being modeled.

Models can be used for various purposes. Thus we distinguish between *descriptive* and *prescriptive* models [17]. *Descriptive models* show a part of the reality to make a specific aspect easier to understand. For example, a city map describes a city in such a way as to help a non-local person to find routes within the city. In contrast, *prescriptive models* are used to offer a construction manual for the system to be developed.

In this book, we look at how the different aspects of a software system can be modeled using a modeling language—the Unified Modeling Language—such that executable code can be derived either manually or (semi)automatically, or easily understandable documentation can be

*Executable code as
model*

created. Incidentally, the executable code, developed in any programming language, such as Java, is also a model. This model represents the problem to be solved and is optimized for execution on computers.

To summarize, there are three applications for models [19]:

- Models as a sketch
- Models as a blueprint
- Models as executable programs

Models are used as a *sketch* to communicate certain aspects of a system in a simple way. Here, the model is not a complete mapping of the system. Sketches are actually distinguished by their selectivity, as they are reduced to the essential aspects for solving a problem. Sketches often make alternative solutions visible. These are then discussed in the development team. Thus, models are also used as a basis for discussion.

Models as a sketch

In contrast to the use of models as sketches, completeness is very important when models are used as a *blueprint*. These models must contain sufficient detail to enable developers to create ready-to-run systems without having to make design decisions. Models used as blueprints often do not specify the whole system, only certain parts. For example, the interface definitions between subsystems are defined in the model, whereby the developers are free to decide on the internal implementation details. If the models are behavioral descriptions, the behavior can also be simulated and tested to identify faults in advance.

Models as a blueprint

Models as sketches and blueprints can be used for both *forward engineering* and *backward engineering*. In forward engineering, the model is the basis for creating code, while in backward engineering, the model is generated from the code to document the code in a clear and easily understandable way.

Forward and backward engineering

Finally, models can be used as *executable programs*. This means that models can be specified so precisely that code can be generated from them automatically. In the context of UML, model-based software development has become extremely popular in recent years; it offers a process for using UML as a programming language. We will address this briefly in Chapter 9 of this book, after we have discussed the basics of UML. In some application areas, such as the development of embedded systems, models are already being used instead of traditional programming languages. In other areas, active research is taking place to raise the development of software systems to a new and more easily maintainable and less error-prone abstraction level.

Models as executable programs

1.3 Object Orientation

Object orientation

If we want to model in an object-oriented style, we must first clarify what *object orientation* means. The introduction of object orientation dates back to the 1960s when the simulation language SIMULA [24] was presented, building on a paradigm that was as natural to humans as possible to describe the world. The object-oriented approach corresponds to the way we look at the real world; we see it as a society of autonomous individuals, referred to as objects, which take a fixed place in this society and must thereby fulfill predefined obligations.

There is not only one single definition for object orientation. However, there is a general consensus about the properties that characterize object orientation. Naturally, objects play a central role in object-oriented approaches. Viewed simply, objects are elements in a system whose data and operations are described. Objects interact and communicate with one another. In general, we expect the concepts described below from an object-oriented approach.

1.3.1 *Classes*

Class

In many object-oriented approaches, it is possible to define *classes* that describe the attributes and the behavior of a set of objects (the instances of a class) abstractly and thus group common features of objects. For example, people have a name, an address, and a social security number. Courses have a unique identifier, a title, and a description. Lecture halls have a name as well as a location, etc. A class also defines a set of permitted operations that can be applied to the instances of the class. For example, you can reserve a lecture hall for a certain date, a student can register for an exam, etc. In this way, classes describe the behavior of objects.

1.3.2 *Objects*

Object

The instances of a class are referred to as its *objects*. For example, `lh1`, the Lecture Hall 1 of the Vienna University of Technology, is a concrete instance of the class `LectureHall`. In particular, an object is distinguished by the fact that it has its own identity, that is, different instances of a class can be uniquely identified. For example, the beamer in Lecture Hall 1 is a different object to the beamer in Lecture Hall 2, even

if the devices are of the same type. Here we refer to *identical* devices but not the *same* device. The situation for concrete values of data types is different: the number 1, which is a concrete value of the data type `Integer`, does not have a distinguishable identity.

An object always has a certain state. A state is expressed by the values of its attributes. For example, a lecture hall can have the state *occupied* or *free*. An object also displays behavior. The behavior of an object is described by the set of its operations. Operations are triggered by sending a message.

1.3.3 Encapsulation

Encapsulation is the protection against unauthorized access to the internal state of an object via a uniquely defined interface. Different levels of visibility of the interfaces help to define different access authorizations. Java, for example, has the explicit visibility markers `public`, `private`, and `protected`, which respectively permit access for all, only within the object, and only for members of the same class, its subclasses, and of the same package.

Encapsulation

1.3.4 Messages

Objects communicate with one another through *messages*. A message to an object represents a request to execute an operation. The object itself decides whether and how to execute this operation. The operation is only executed if the sender is authorized to call the operation—this can be regulated in the form of visibilities (see the previous paragraph)—and a suitable implementation is available. In many object-oriented programming and modeling languages the concept of *overloading* is supported. This enables an operation to be defined differently for different types of parameters. For example, the operator `+` realizes different behavior depending on whether it is used to add up integers (e.g., `1 + 1 = 2`) or to concatenate character strings (e.g., `"a" + "b" = "ab"`).

Message

Overloading

1.3.5 Inheritance

The concept of *inheritance* is a mechanism for deriving new classes from existing classes. A subclass derived from an existing class (= su-

Inheritance

perclass) inherits all visible attributes and operations (specification and implementation) of the superclass. A subclass can:

- Define new attributes and/or operations
- Overwrite the implementation of inherited operations
- Add its own code to inherited operations

Class hierarchy

Inheritance enables extensible classes and as a consequence, the creation of *class hierarchies* as the basis for object-oriented system development. A class hierarchy consists of classes with similar properties, for example, $\text{Person} \leftarrow \text{Employee} \leftarrow \text{Professor} \leftarrow \dots$ where $A \leftarrow B$ means that B is a subclass of A.

When used correctly, inheritance offers many advantages: reuse of program or model parts (thus avoiding redundancy and errors), consistent definition of interfaces, use as a modeling aid through a natural categorization of the occurring elements, and support for incremental development, i.e., a step-by-step refinement of general concepts to specific concepts.

1.3.6 Polymorphism

Polymorphism

In general terms, *polymorphism* is the ability to adopt different forms. During the execution of a program, a polymorphic attribute can have references to objects from different classes. When this attribute is declared, a type (e.g., class `Person`) is assigned statically at compile time. At runtime, this attribute can also be bound dynamically to a subtype (e.g., subclass `Employee` or subclass `Student`).

A polymorphic operation can be executed on objects from different classes and have different semantics in each case. This scenario can be implemented in many ways: (i) via *parametric polymorphism*, better known as genericity—here, type parameters are used. In Java for example, the concrete classes are transferred to the operations as arguments; (ii) via *inclusion polymorphism*—operations can be applied to classes and to their direct and indirect subclasses; (iii) via *overloading of operations*; and (iv) via *coercion*, that is, the conversion of types. The first two methods above are known as *universal polymorphism*; the other two methods are referred to as *ad hoc polymorphism* [13].

1.4 The Structure of the Book

In Chapter 2 we give a short overview of UML by recapitulating the history of its creation and taking a brief look at its 14 different diagrams. Then, in Chapter 3, we introduce the concepts of the use case diagram. This diagram enables us to describe the requirements that a system to be developed should satisfy. In Chapter 4 we present the class diagram. This diagram allows us to describe the structure of a system. To enable us to model the behavior of a system, in Chapter 5 we introduce the state machine diagram, in Chapter 6 the sequence diagram, and in Chapter 7 the activity diagram. We explain the interaction of the different types of diagrams in Chapter 8 with three examples. In Chapter 9, we briefly examine advanced concepts that are of significant importance for the practical use of UML.

The concepts are all explained using examples, all of which are based on the typical Austrian university environment. In most cases they represent heavily simplified scenarios. It is not our intention in this book to model one single, continuous system, as there is a high risk that in doing so we would become lost in a multitude of technical details. We have therefore selected examples according to their didactic benefit and their illustrative strength of expression. In many cases, we have therefore made assumptions that, for didactic reasons, are based on simplified presentations of reality.

UML is based entirely on object-oriented concepts. This is particularly noticeable in the class diagram, which can easily be translated into an object-oriented programming language. We will get to know the class diagram and possible translations to program code in Chapter 4. However, UML has not been designed for one specific object-oriented language. For the sake of readability, we use a notion of object-orientation as found in modern programming languages like Java or C#.

Chapter 2

A Short Tour of UML

Before introducing the most important concepts of UML in the following chapters, we first explain the background of this modeling language. We look at how UML came into being and what the “U” for “Unified” actually means. We then answer the question of how UML itself is defined, that is, where do the rules come from that dictate what a valid model should look like in UML? Furthermore, we outline what UML is used for. Finally, we give a short overview of all 14 UML diagrams in the current version 2.4.1 of the UML standard specification. These diagrams can be used for modeling both structure and behavior.

2.1 The History of UML

The introduction of object-oriented concepts in information technology originates from the work of the early 1960s [12]. The first ideas were implemented in systems such as Sketchpad, which offered a new, graphical communication approach between man and computer [28, 51].

Today, the programming language SIMULA [24] is regarded as the first object-oriented programming language. SIMULA was primarily used to develop simulation software and was not particularly widely used. It already included concepts such as classes, objects, inheritance, and dynamic binding [2].

The introduction of these concepts was the start of a revolution in software development. In the subsequent decades, there followed a multitude of programming languages based on the object-oriented paradigm [21]. These included languages such as C++ [50], Eiffel [31], and Smalltalk [28]. They already contained many of the important concepts of modern programming languages and are still used today.

Origins of object orientation

SIMULA

Object-oriented programming languages

The emergence and introduction of object orientation as a method in software engineering is closely connected to the appearance of object-oriented programming languages. Today, object orientation is a proven and well-established approach for dealing with the complexity of software systems. It is applied not only in programming languages but also in other areas, such as in databases or the description of user interfaces.

As we have already discussed in Section 1.2, where we introduced the notion of a model, software systems are abstractions aimed at solving problems of the real world with the support of machines. Procedural programming languages are not necessarily the most appropriate tools for describing the real world: the differences in concept between a natural description of a problem and the practical implementation as a program are huge. Object-oriented programming was an attempt to develop better programs that, above all, are easier to maintain [12].

Over the years, object orientation has become the most important software development paradigm. This is reflected in object-oriented programming languages such as Java [4] or C# [32] and object-oriented modeling languages such as UML. However, the road to the current state-of-the-art of software development was long and winding.

Ada In the 1980s, the programming language Ada, funded by the United States Department of Defense, was extremely popular due to its powerful concepts and efficient compilers [25]. Even back then, Ada supported abstract data types in the form of *packages* and concurrency in the form of *tasks*. Packages allowed the separation of specification and implementation and the usage of objects and classes of objects. Ada thus distinguished itself fundamentally from other popular languages of that time, such as Fortran and Cobol. As a consequence, there followed a great demand for object-oriented analysis and design methods to make the development of Ada programs easier. Due to the wide distribution of Ada and the pressure from the United States Department of Defense, these modeling methods were based specifically on the characteristics of Ada. Grady Booch was one of the first researchers to publish work on the object-oriented design of Ada programs [5].

Booch method

Over time, a number of further object-oriented modeling methods arose (see [12] for an overview). In general, the modeling methods had either a strong reference to programming languages, such as the Booch method, or a strong reference to data modeling, such as the *Object Modeling Technique* (OMT) approach developed by James Rumbaugh et al. [42]. OMT supported the development of complex objects in the sense of an object-oriented extension of the entity-relationship model [14] which had been introduced for describing databases.

*OMT approach by
Rumbaugh et al.*

*OOSE approach by
Jacobson et al.*

Independently of this, Ivar Jacobson et al. introduced the *Object-Oriented Software Engineering* (OOSE) approach [27]. This approach was originally developed to describe telecommunication systems.

In the 1980s and early 1990s, the modeling world was flooded with a multitude of different modeling languages. Considerable effort was required to deal with the resulting compatibility problems. The models of different project partners were often not compatible and it was not always possible to reuse models in different projects. The result was exhausting discussions about different notations, which detracted from the actual modeling problems. As new modeling languages were appearing all the time, there was no clarity about which were worthy of investment and which were just a short-lived trend. If a language did not become accepted, all investments that had been made to establish it within a project or a company were generally lost. Looking back, this time of numerous approaches, often with the difference being only in the detail, is referred to as the *method war*.

Method war

To put an end to this unsatisfactory situation, in 1996 the *Object Management Group* (OMG) [33], the most important standardization body for object-oriented software development, called for the specification of a uniform modeling standard.

Object Management Group (OMG)

In the previous year, 1995, Grady Booch, Ivar Jacobson, and James Rumbaugh had combined their ideas and approaches at the OOPSLA conference (OOPSLA stands for Object-Oriented Programming, Systems, Languages, and Applications). Since then, Booch, Jacobson, and Rumbaugh have often been called the “three amigos”. They set themselves the following objectives [1]:

Three amigos

- Use of object-oriented concepts to represent complete systems rather than just one part of the software
- Establishment of an explicit relationship between modeling concepts and executable program code
- Consideration of scaling factors that are inherent in complex and critical systems
- Creation of a modeling language that can be processed by machines but can also be read by human beings

The result of their efforts was the *Unified Modeling Language* (UML) which was submitted in version 1.0 in 1997 in response to the OMG call. A number of former competitors were involved in the creation of version 1.1 that subsequently appeared in 1998. One of the main objectives was a consistent specification of the language core of UML which is documented in the *metamodel* (see Chapter 9). The metamodel defines which model elements the language UML provides and how to use them correctly. For formulating constraints which the model elements have to fulfill, the *Object Constraint Language* (OCL) [36], based on predicate logic, was introduced. In subsequent versions, along with the revision of certain language concepts, mechanisms for the interchangeability of models in the form of the *XML Metadata Interchange format*

Unified Modeling Language (UML)

Metamodel

Object Constraint Language (OCL)

*XML Metadata
Interchange format
(XMI)*

(XMI) [38] were added. In addition to these rather small changes, in 2000 the OMG initiated a modernization process for UML. This finally led to the adoption of the language standard UML 2.0 in 2005. With the exception of small changes which, through interim versions, resulted in the current version 2.4.1, this is the language description of UML that we will get to know and use in this book.

Today, UML is one of the most widespread graphical object-oriented modeling languages. Despite the numerous revisions, its roots (Booch method, OMT, OOSE) are still clearly recognizable. UML is suitable for modeling both complex object relationships and processes with concurrency. UML is a general purpose modeling language, meaning that its use is not restricted to a specific application area. It provides language and modeling concepts and an intuitive graphical notation for modeling various application areas, enabling a software system to be specified, designed, visualized, and documented [43]. The result of modeling with UML is a graphical model that offers different views of a system in the form of various diagrams.

2.2 Usage

UML is not tied to a specific development tool, specific programming language, or specific target platform on which the system to be developed must be used. Neither does UML offer a software development process. UML in fact separates the modeling language and modeling method. The latter can be defined on a project-specific or company-specific basis. However, the language concepts of UML do favor an iterative and incremental process [43].

*Use in the software
development process*

UML can be used consistently across the entire software development process. At all stages of development, the same language concepts can be used in the same notation. Thus, a model can be refined in stages. There is no need for a model to be translated into another modeling language. This enables an iterative and incremental software development process. UML is well-suited for various application areas with different requirements regarding complexity, data volume, real time, etc.

*Generic language
concepts*

The UML model elements and their correct use are specified in the UML *metamodel* [35]. The language concepts are defined so generically that a wide and flexible applicability is achieved. To avoid restricting the use of UML, the standard is (intentionally) vague at various points, permitting different interpretations in the form of semantic variation points. However, this is a two-edged sword; it also leads to different implementations of the language standard by modeling tools, which in turn, unfortunately makes it difficult to exchange models.

Semantic variation point

2.3 Diagrams

In UML, a model is represented graphically in the form of *diagrams*. A diagram provides a view of that part of reality described by the model. There are diagrams that express which users use which functionality and diagrams that show the structure of the system but without specifying a concrete implementation. There are also diagrams that represent supported and forbidden processes. In the current version 2.4.1, UML offers 14 diagrams that describe either the structure or the behavior of a system.

Diagram

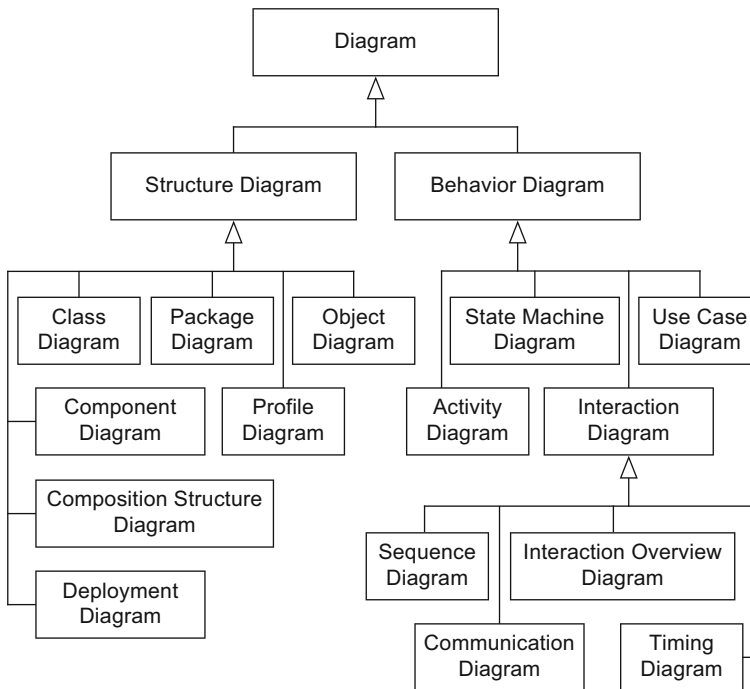


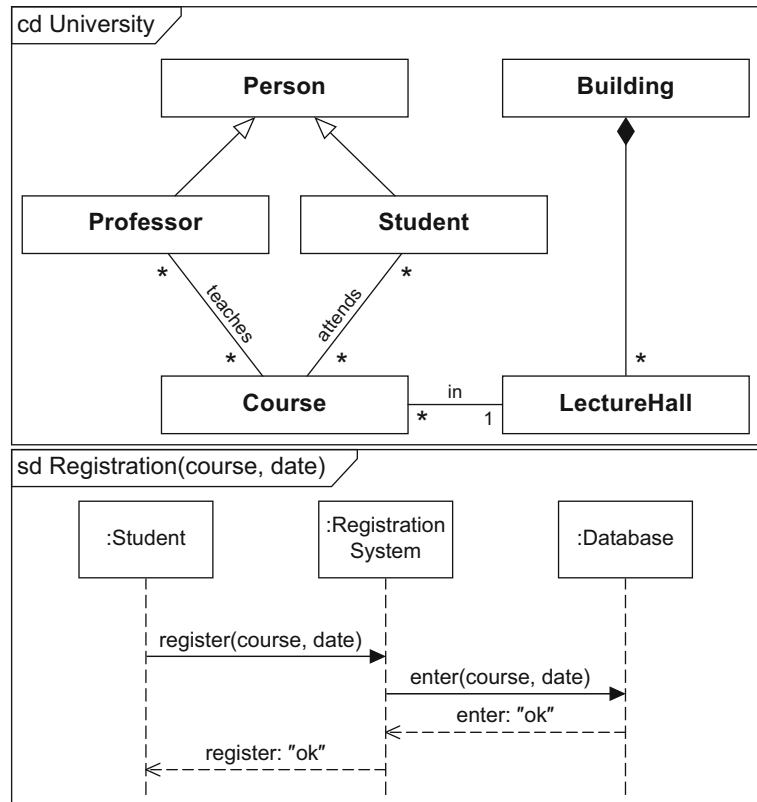
Figure 2.1
UML diagrams

Figure 2.1 shows a taxonomy of the 14 UML diagrams [35], giving a very rough categorization. As the figure shows, we differentiate between *structure diagrams* and *behavior diagrams*. The behavior diagrams include the interaction diagrams, which in turn consist of four diagrams (see Chapter 6).

A diagram is usually enclosed by a rectangle with a pentagon in the top left-hand corner. This pentagon contains the diagram type and the name of the diagram. Optionally, parameters may be specified following the name which then can be used within the diagram. Figure 2.2 con-

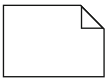
*Notation for diagram
frame*

Figure 2.2
Examples of UML diagram
frames



tains two examples of diagram frames. In particular, it shows a class diagram (cd) with the name University and a sequence diagram (sd) called Registration with the parameters course and date.

Note



A concept that may occur in all diagrams is the *note*. A note can contain any form of expression that specifies the diagram and its elements more precisely—for example, in natural language or in the Object Constraint Language (OCL). Notes may be attached to all other model elements. Figure 2.3 shows an example of the use of a note which specifies in natural language that persons are not permitted to grade themselves. The class Person and the association grades represent concepts of the class diagram that will be introduced in Chapter 4.

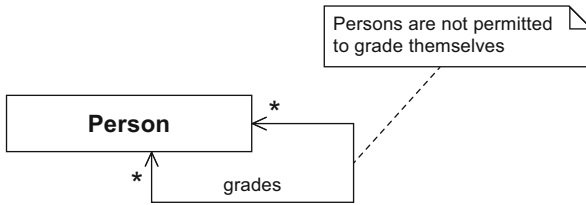


Figure 2.3
Example of a note

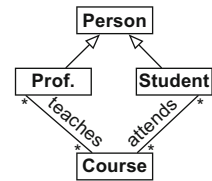
2.3.1 Structure Diagrams

UML offers seven types of diagrams for modeling the structure of a system from different perspectives. The dynamic behavior of the elements in question (i.e., their changes over time) is not considered in these diagrams.

The Class Diagram

Just like the concepts of the object diagram (see next paragraph), the concepts of the *class diagram* originate from conceptual data modeling and object-oriented software development. These concepts are used to specify the data structures and object structures of a system. The class diagram is based primarily on the concepts of *class*, *generalization*, and *association*. For example, in a class diagram, you can model that the classes *Course*, *Student*, and *Professor* occur in a system. Professors teach courses and students attend courses. Students and professors have common properties as they are both members of the class *Person*. This is expressed by a generalization relationship.

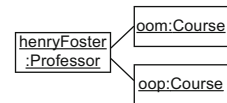
Class diagram
(see Chapter 4)



The Object Diagram

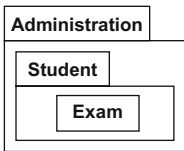
Based on the definitions of the related class diagram, an *object diagram* shows a concrete snapshot of the system state at a specific execution time. For example, an object diagram could show that a professor Henry Foster (henryFoster) teaches the courses Object-Oriented Modeling (oom) and Object-Oriented Programming (oop).

Object diagram
(see Chapter 4)



The Package Diagram

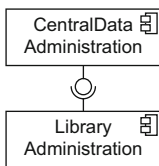
Package diagram



The *package diagram* groups diagrams or model elements according to common properties, such as functional cohesion. For example, in a university administration system, you could introduce packages that contain information about the teaching, the research, and the administrative aspects. Packages are often integrated in other diagrams rather than being shown in separate diagrams.

The Component Diagram

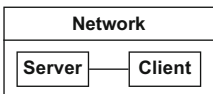
Component diagram



UML pays homage to component-oriented software development by offering *component diagrams*. A component is an independent, executable unit that provides other components with services or uses the services of other components. UML does not prescribe any strict separation between object-oriented and component-oriented concepts. Indeed, these concepts may be combined in any way required. When specifying a component, you can model two views explicitly: the external view (black box view), which represents the specification of the component, and the internal view (white box view), which defines the implementation of the component.

The Composition Structure Diagram

Composition structure diagram



The *composition structure diagram* allows a hierarchical decomposition of the parts of the system. You can therefore use a composition structure diagram to describe the internal structure of classes or components in detail. This enables you to achieve a higher level of detail than, for example, in a class diagram because the modeling is context-specific. You can specify details of the internal structure that are valid precisely for the context under consideration.

The Deployment Diagram

Deployment diagram

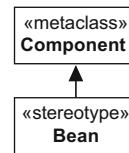


The hardware topology used and the runtime system assigned can be represented by the *deployment diagram*. The hardware encompasses processing units in the form of nodes as well as communication relationships between the nodes. A runtime system contains artifacts that are deployed to the nodes.

The Profile Diagram

Using *profiles*, you can extend UML to introduce domain-specific concepts. The actual core of the language definition of UML, the meta-model, remains unchanged. You can thus reuse modeling tools without having to make adjustments. For example, you can use profiles to introduce the concept of Java Enterprise Beans.

Profile diagram



2.3.2 Behavior Diagrams

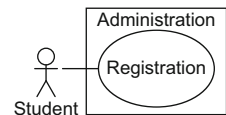
With the *behavior diagrams*, UML offers the infrastructure that enables you to define behavior in detail.

Behavior refers to the direct consequences of an action of at least one object. It affects how the states of objects change over time. Behavior can either be specified through the actions of a single object or result from interactions between multiple objects.

The Use Case Diagram

UML offers the *use case diagram* to enable you to define the requirements that a system must fulfill. This diagram describes which users use which functionalities of the system but does not address specific details of the implementation. The units of functionality that the system provides for its users are called *use cases*. In a university administration system, for example, the functionality Registration would be a use case used by students.

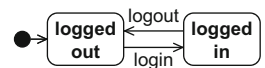
Use case diagram
(see Chapter 3)



The State Machine Diagram

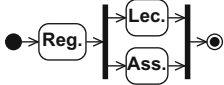
Within their life cycle, objects go through different states. For example, a person is in the state logged out when first visiting a website. The state changes to logged in after the person successfully entered username and password (event login). As soon as the person logs out (event logout), the person returns to the state logged out. This behavior can be represented in UML using the *state machine diagram*. This diagram describes the permissible behavior of an object in the form of possible states and state transitions triggered by various events.

State machine diagram
(see Chapter 5)



The Activity Diagram

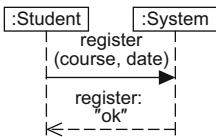
Activity diagram
(see Chapter 7)



You can model processes of any kind using *activity diagrams*: both business processes and software processes. For example, an activity diagram can show which actions are necessary for a student to participate in a lecture and an assignment. Activity diagrams offer control flow mechanisms as well as data flow mechanisms that coordinate the actions that make up an activity, that is, a process.

The Sequence Diagram

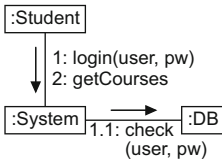
Sequence diagram
(see Chapter 6)



The *sequence diagram* describes the interactions between objects to fulfill a specific task, for example, registration for an exam in a university administration system. The focus is on the chronological order of the messages exchanged between the interaction partners. Various constructs for controlling the chronological order of the messages as well as concepts for modularization allow you to model complex interactions.

The Communication Diagram

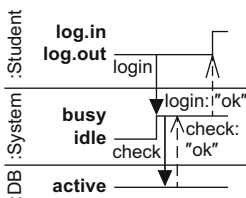
Communication diagram
(see Chapter 6)



Similarly to the sequence diagram, the *communication diagram* describes the communication between different objects. Here, the focus is on the communication relationships between the interaction partners rather than on the chronological order of the message exchange. Complex control structures are not available. This diagram clearly shows who interacts with whom.

The Timing Diagram

Timing diagram
(see Chapter 6)

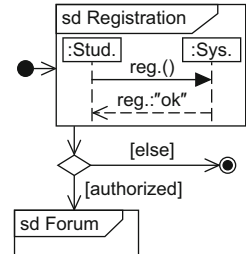


The *timing diagram* explicitly shows the state changes of the interaction partners that can occur due to time events or as a result of the exchange of messages. For example, a person is in the state logged in as soon as the message is received from the university administration system that the password sent is valid.

The Interaction Overview Diagram

The *interaction overview diagram* models the connection between different interaction processes by setting individual interaction diagrams (i.e., sequence diagram, communication diagram, timing diagram, and other interaction overview diagrams) in a time-based and causal sequence. It also specifies conditions under which interaction processes are permitted to take place. To model the control flow, concepts from the activity diagram are used. For example, a user of the university administration system must first log in (which already represents a separate interaction with the system) before being allowed to use further functionalities.

Interaction overview diagram
(see Chapter 6)



2.4 Diagrams Presented in this Book

As already explained in Chapter 1, this book restricts itself to the five most important and most widespread types of UML diagrams, namely the use case diagram, class diagram (including the object diagram), state machine diagram, sequence diagram, and activity diagram. In this book, we present these diagrams in the order in which they would generally be used in software development projects. We begin with the use case diagram, which specifies the basic functionality of a software system. The class diagram then defines which objects or which classes are involved in the realization of this functionality. The state machine diagram then defines the intra-object behavior, while the sequence diagram specifies the inter-object behavior. Finally, the activity diagram allows us to define those processes that “implement” the use cases from the use case diagram.